Name: Shimon Kumar Bhandari

ID: 005030903

Course: Algorithms and Data Structures (MSCS-532-A01)

Assignment 3

Randomized Quicksort Analysis

Analysis:

Randomized Quick sort selects random element as pivot in all of the recursive calls. It then partitions the array in two subarrays where all the elements smaller than the pivot element are on the left side and all the elements that are greater than the pivot element are on the right side. By applying this process recursively on smaller subarrays on each call, it sorts the array. Recursive partition size determines the primary factor affecting time complexity measurement.

Time Complexity:

The time complexity of Randomized Quicksort depends on Partitioning the array. During partitioning the algorithm spends O(n) time to traverse the array then moves elements into proper subarray based on the pivot position. The complexity of time also depends on Recursive calls. The selection of pivot element determines the size of subproblems.

Let T(n) be time taken to sort n size array.

$$T(n)=T(i)+T(n-i-1)+O(n)$$

where,

i is the number of elements in left subarray

$n-i-1$ is the number of elements in right subarray

O(n) is the cost to partition the array

It is equally likely that the random pivot quicksort will partition the array into sub array of any ratio. Below is the proof by substitution method that the average time complexity for randomized Quicksort is O(nlogn).

Substitution method:

Assume T(n)=cnlogn and substitute into the recurrence:

As we know, the average and best case for Quick sort is:

T(n)=2T(n/2)+cn

=2(c(n/2)log(n/2))+cn

=cnlogn−cn+cn

=cnlogn

This satisfies our recurrence. Therefore, T(n)=O(nlogn) is the average case time complexity of randomized quick sort.

Comparison:

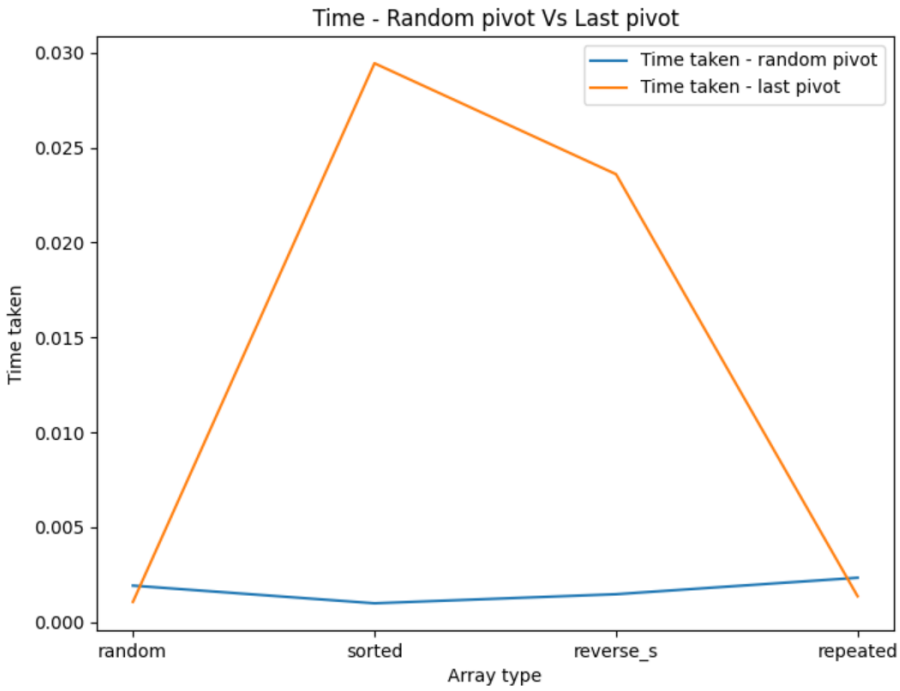

Fig: Time comparison of randomized Quicksort with Last element Quicksort

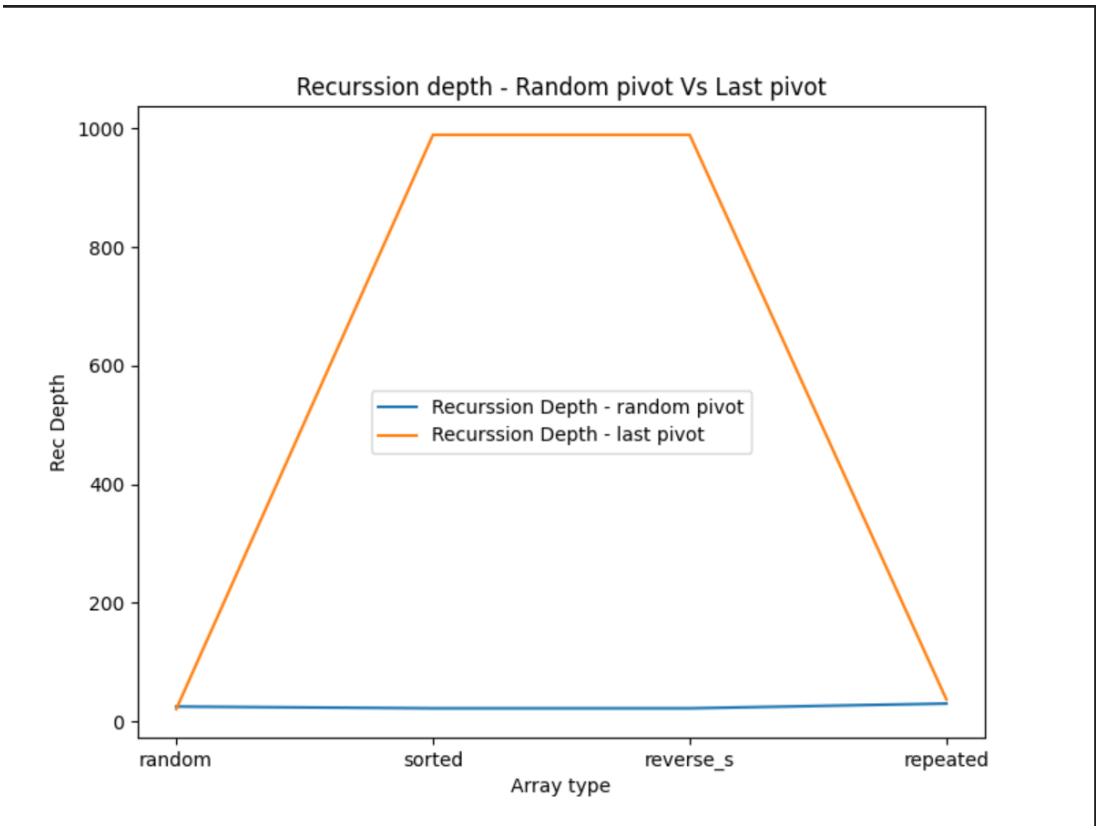

Fig: recursion Depth comparison of randomized Quicksort with Last element Quicksort

As we can see above, the time taken by the randomized Quicksort is always similar against all types of arrays (Randomly generated array, already sorted arrays, Reverse-sorted arrays, Arrays with repeated elements). But the time taken by the last element pivot Quicksort is different for different types of arrays. It does worst for already sorted and reverse sorted array. This is because the subarrays at each recursion do not have any element in one side of pivot.

The same goes for space complexity, for randomized Quicksort the maximum depth of recursion is always similar for all types of arrays. But for the last element pivot Quicksort, the recursion depth is equal to the size of array for reverse sorted and sorted array. This is also because there are no elements on one side of pivot at each recursive call.

Therefore, randomized Quicksort performs better than the last element as pivot Quicksort.

Hashing with Chaining

Hash table with uniform hashing achieves $O(1+\alpha)$ time complexity for search, insert and delete operations based on the load factor (α) which is the ratio of **n** elements stored to **m** table slots available. The time required to locate a bucket is constant ($O(1)$ but traversing through bucket lists depends on the load factor $O(\alpha)$.

Performance strongly relies on how the load factor influences operations within the system. Operations performed on hash tables with low load factors ($\alpha \approx 0$) exhibit faster performance because fewer elements reside in each bucket, but this approach leads to high memory consumption. Extremely high load factors ($\alpha > 1$) result in longer bucket chains that decrease performance because it takes longer to traverse these lengthy sequences. Hash tables reach their best operational memory efficiency when their load factor stabilizes between 0.5 and 1.

The correct management of load factor alongside collision reduction techniques requires multiple strategies implementation. Dynamic resizing techniques increase a hash table's size by a factor of two whenever the specified load factor threshold is surpassed. The process of resizing causes elements to be rehashed before they receive new positions on the expanded table to maintain a balanced key distribution. The $O(n)$ resizing cost persists only as a one-time expense that spreads across multiple operations thus maintaining low average time complexity. Hash function selection becomes essential for performance by using universal hashing-based methods which distribute keys evenly while preventing clustering effects. When tables use large prime numbers as their sizes, they minimize the chances of hash value patterns creating collisions.

Resources:

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.

Goodrich, M. T., & Tamassia, R. (2013). *Data Structures and Algorithms in Java*. Wiley.